University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Spring 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming deadlines:
  - Homework 3 is due on 2/7
  - Homework 4 is due on 2/14
  - Lab 3 is due on 2/11

Previous lecture …

- How to add and delete from a Binary Search Tree (BST)
- Red-Black BST

CourseMIRROR Reflections

This Lecture

- How to add and delete from a Red-Black BST
- How to traverse a Binary Tree

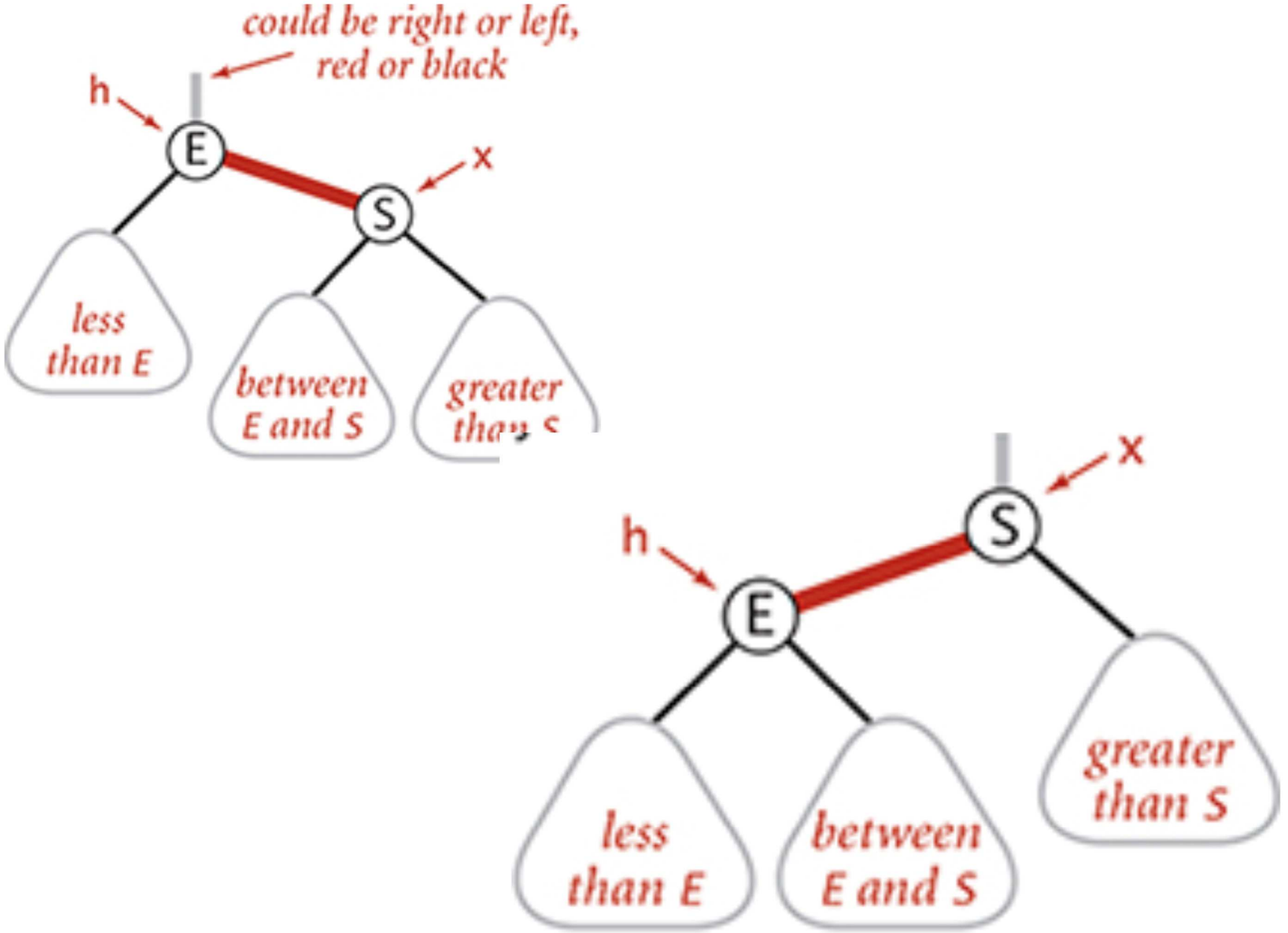
Red-Black BST

- Definition
  - two colors for links (nodes)
    - a node has the same color as the link to its parent
  - root node is always black
  - null leaves are always black
  - red links are always to the left children
  - at most one red-link per node
  - all root-to-null-leaf paths have the same number of black nodes
  - ***Why?***
    - ***maximum height = 2*log n !!***
- Basic operations
  - rotate left
  - rotate right
  - flip color
  - ***preserve the properties of the red-black BST!***

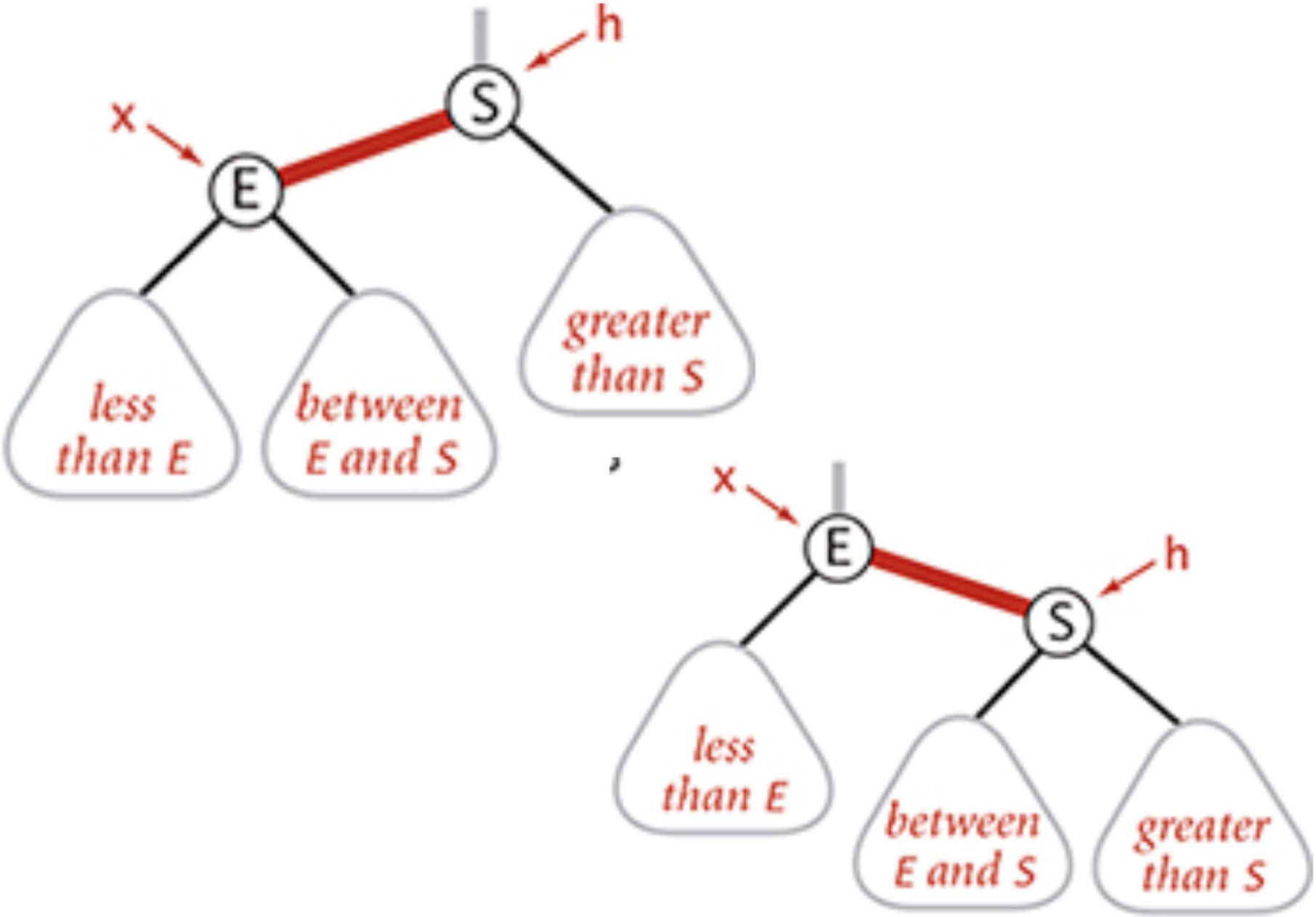
Adding to a RB-BST

- Ok, so we add a red leaf node!
- What can go wrong then?
  - The new node is a right child
  - The parent of the new node is also red
  - The sibling of the new node is also red

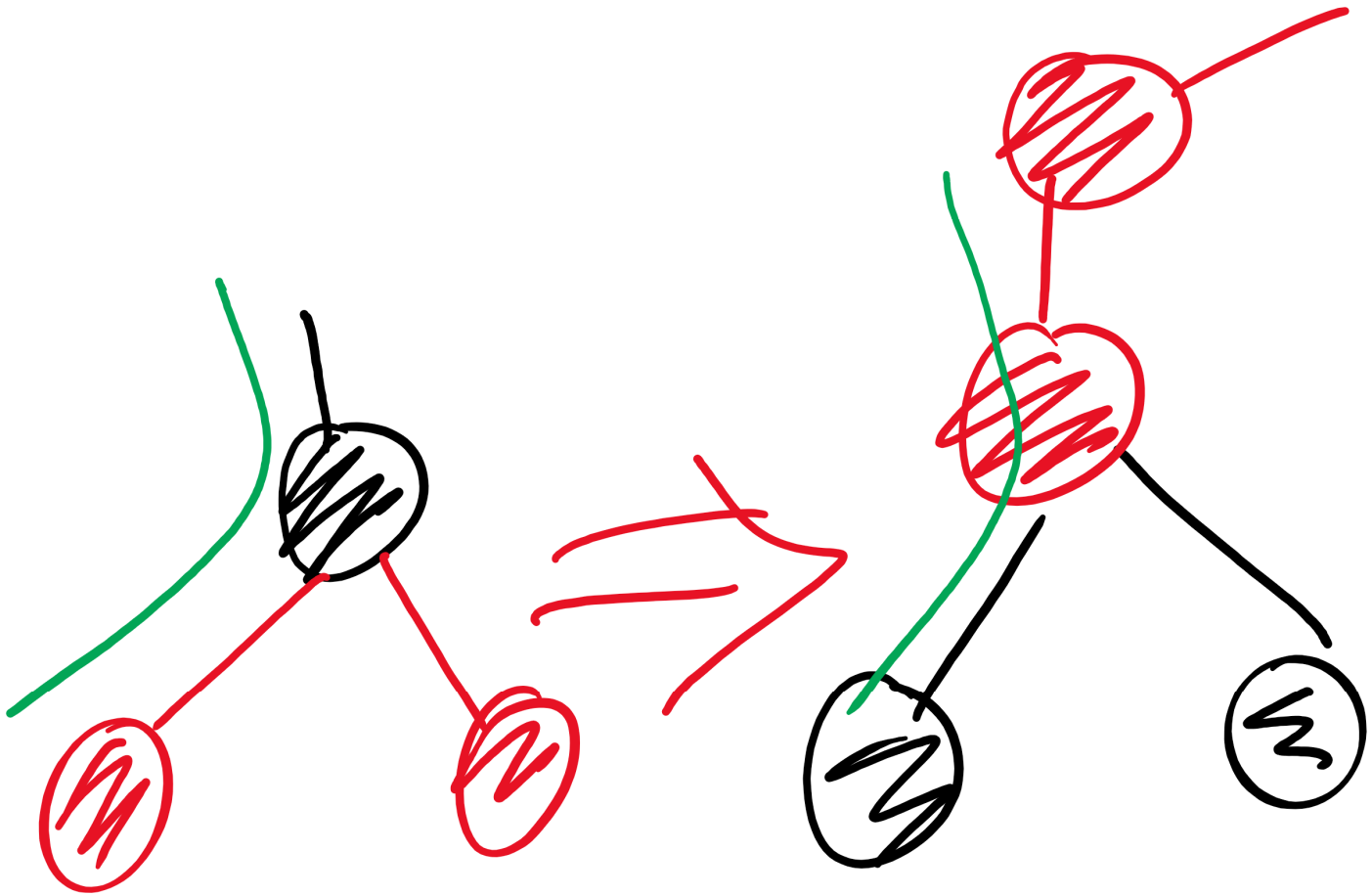
What if the new red node is a right child?



rotateRight



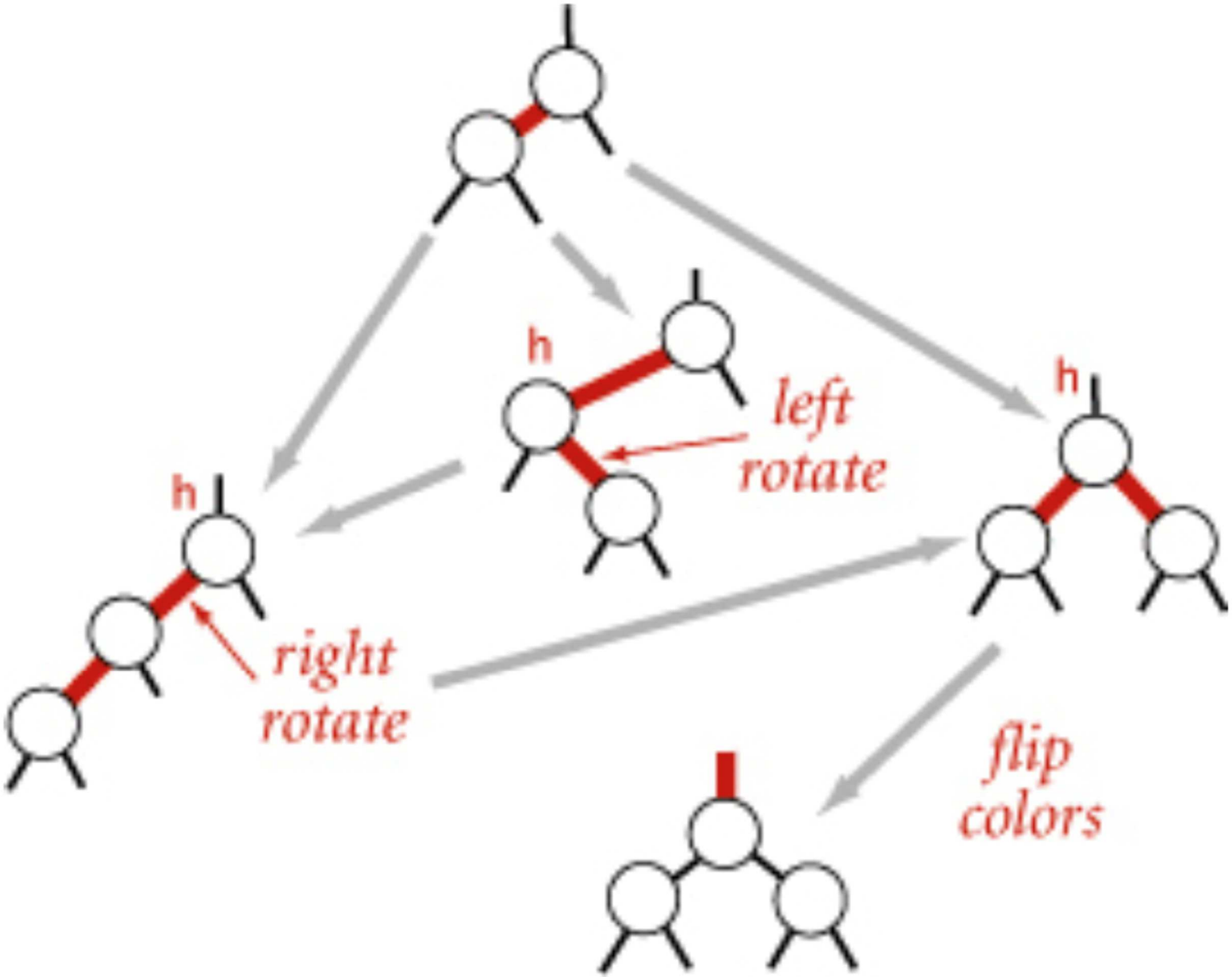
colorFlip



Adding to a red-black BST

- new node is always red (at least initially)
- if properties **violated**, correct using one or more of the basic operations
- Violations that can happen:
  - red link to the right child
  - two red links connected to the same node
  - root node is red
- Correcting a violation may result in a violation up the tree
- Corrections happen as we climb back up the tree
  - That is, ***after the recursive call***

Dependencies between corrections!



Which violations to check for first?

```
1  // insert the key-value pair in the subtree rooted at h
2      private Node put(Node h, Key key, Value val) {
3          if (h == null) return new Node(key, val, RED, 1);
4
5          int cmp = key.compareTo(h.key);
6          if      (cmp < 0) h.left  = put(h.left,  key, val);
7          else if (cmp > 0) h.right = put(h.right, key, val);
8          else              h.val   = val;
9
10         // fix-up any right-leaning links
11         if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
12         if (isRed(h.left)  &&  isRed(h.left.left)) h = rotateRight(h);
13         if (isRed(h.left)  &&  isRed(h.right))     flipColors(h);
14         h.size = size(h.left) + size(h.right) + 1;
15
16         return h;
17     }
```

Deleting a node

- Make sure that we are not deleting a black node
  - as we go down the tree, make sure that the next node down is red
    - using certain operations
  - as we go back up the tree, correct any violations
    - same as we did while adding
  - if deleting a node with 2 children
    - replace with minimum of right subtree
    - similar trick to delete in regular BST

Other operations

- rank, select, range query, …
- Same code as regular BST!
- **worst-case runtime = Theta(log n)!!**

Please submit your reflections by using the CourseMIRROR App

If you are having a problem with CourseMIRROR, please send an email to coursemirror.development@gmail.com